Akimot User Manual

Tomáš Kozelek

2009

0.1 About

Akimot is an engine for the game of Arimaa. The very core function of the program is to take a valid Arimaa position and produce a suggested move. The program is command line oriented, highly configurable and supporting communication in both traditional *getMove interface* and relatively new *Arimaa Engine Interface*. There are several support applications delivered together with the program. The whole project is distributed under the GNU GPL license.

0.2 Background

The program is written in C++ programming language. The target platform are the *Linux* systems. Building the program at *Windows* machines hasn't been tested. For development we used the *Vim* integrated development environment together with *Scons* software build system. As a source version control tool we used the *Git* software. As a unit testing module we used the *cpptest* library. We used *gdb* for debugging and *gprof* for profiling. Moreover we programmed several external testing tools. For instance to verify that our search extension for finding goal works properly we harvested all the positions where rabbit can score a goal from all the games ever played online. And through *AEI* we tested the ability of our program to find the goal moves in these positions (> 50000).

0.3 Installation

Project is distributed as a snapshot of the development version. This snapshot is included on the attached CD. The rough organization of the project is following:

akimot

source files - **.h*, **.cpp*

a - shortcut for `akimot -e -a init`

AUTHORS - authors information

COPYING - license information

default.cfg - an example akimot's configuration file

doc - Doxygen generated reference documentation

Doxy - configuration file for documentation generator

init - file with initial commands for the AEI session with the program

INSTALL - installation and compilation instructions

other - Support software

aei - Arimaa Engine Interface source codes

match - Match environment with example bots

ats - Arimaa Test Suite code and tests

tagui - Development GUI

rabbits - Large scale goal check unit

aga - Tools for downloading and filtering the games from online archive

paths.py - small support file with paths definition

s - shortcut for `scons opt=1`

SConstruct - configuration file for the build process

TODO - programming issues TODO list

There are no precompiled binaries therefore the program itself must be built from the source codes. Recommended way for building the binary is to use a *Scons* tool. There is a prepared *SConstruct* configuration file for this job. Following commands might be used :

`scons` development build

`scons opt=1` optimized build (**recommended for standard use**)

`scons prof=1` build including profiling information

`scons dbg=1` build for debugging purpose

`scons -c` clean the build (removes object files and binary)

`scons cfg=1` copies configuration file *default.cfg* to predefined places (e.g. *match* dir, *ats* dir, etc.)

The built program should get automatically installed to the proper destinations in subsequent directories (e.g. **match/bot_akimot**, **ats/bot_akimot**, etc.). You can force the installation as well - for instance to force installation of optimized source to **match/bot_akimot** just issue `scons opt=1 match`.

0.4 Options and Configuration

Command line options are used to customize the general behavior (mode the program is in, communication protocol to use, etc.). The syntax for running the program is: `akimot [options] [position_file [game_record_file]]`. The files listed after options are part of prescribed communication via the *getMove interface*. The options are following:

`-h` prints small help

`-b` runs benchmarks

`-e` uses extended AEI command set

`-a file` uses given file to init the AEI session

`-g` runs in getMove mode (`position_file` and/or `game_record_file` must be supplied then)

`-c file` uses given file as a configuration file

Configuration file is used to modify the properties of the search engine. The default configuration file named `default.cfg` is present in the project's root. This configuration file is used by the program if not specified otherwise in command line options. Moreover, it contains the "best" configuration used for scalability tests and games in the gameroom. Every item in the file is documented and should be easy to understand. Various issues might be influenced in the configuration file, for instance:

- details of the algorithm (time settings, exploration coefficient in UCT, mature level, length of the playouts, etc.)
- various (on/off) extensions (transposition tables, knowledge bias in playouts, search extensions, etc.)
- parallelism degree (number of threads to use)
- weights of the evaluation method (e.g. what are particular pieces worth, how are traps evaluated, what is camel hostage penalty etc.)
- weights of the step evaluation

Little effort was invested to make the program's input processing "dummy proof".

0.5 Session

Program supports two distinct ways of interaction:

getMove mode

So far, this communication protocol has been recognized as an official protocol for computer challenge Arimaa championships. Program takes three files with *game_position*, *game_record* and *game_state* and outputs a move to be made. For a long time this was an only way how to connect a bot to the online gameroom. In Akimot this mode must be explicitly toggled with `-g` option, moreover not all three files must be present (actually information from *game_state* file are not used by Akimot at all). Example sessions:
load from the position

```
tomik@linda ~/src/akimot $./akimot -g data/captures/02.ari
Ed2n Ed3n Ed4n Ed5n
```

load from the record

```
tomik@linda ~/src/akimot $./akimot -g data/captures/02.are
Ed2n Ed3n Ed4n Ed5n
```

load from the record (preferred over the position)

```
tomik@linda ~/src/akimot $
./akimot -g data/captures/02.ari data/captures/02.are
Ed2n Ed3n Ed4n Ed5n
```

AEI mode

Akimot implements the textual AEI protocol as described below. This is a preferred way of communication with the program. We also used this model for a connection to the gameroom. Example session[1] :

```
tomik@linda ~/src/akimot $ ./akimot -e
#start the initial opening phase (a handshake)
<aei
>id name akimot
>id author Tomas Kozelek
>id version 0.1
>aeiok
#handskake was performed successfully now ping the engine
<isready
>readyok
#start the new game
<newgame
#set the time for move per sec to 5
<setoption name tcmove value 5
<setpositionfile data/captures/02.ari
```

[1]symbol <and >are used only here to emphasize the direction of communication, # marks a comment

```
<go
>log Debug: Search finished. Suggested move: Ed2n Ed3n Ed4n Ed5n
>info stat UCT:
>info stat   355069 playouts
>info stat   4.90003 seconds
>info stat   72462 playouts per second
>info stat   449253 nodes in the tree
>info stat   23326 nodes expanded
>info stat   77962 nodes pruned
>info stat   6.76431 average descends in playout
>info stat   best move: Ed2n Ed3n Ed4n Ed5n
>info stat   best move visits: 113535
>info stat   win condidence: 0.436083
>info time 4.90004
>info winratio 0.436083
>bestmove Ed2n Ed3n Ed4n Ed5n
>log Info: over
<quit
>log Info: bye
```

Akimot allows to init the *AEI* session with user defined commands. Commands are written to the init file and given to the program on the command line. Only the beginning of the session (e.g. `aeiinit`, `newgame` command, etc.) or the whole session as well might be issued. In the case of whole session being performed from the init file some commands from AEI extended set must be used(i.e. `gonothread` instead of `go` otherwise following commands are sent to the engine too early). Example init file called *init* with comments (`akimot -e -a init` runs the engine with this init file in extended AEI mode):

```
aei
newgame
setoption name tcmove value 6
setpositionfile data/captures/06.ari
gonothread
#view the board and the resulting tree
boarddump
treedump
#make the move proposed by the previous (recent) search
makemoverec
#start new search(now from the other player's view)
gonothread
quit
```

0.5.1 Position formats

Current position might be communicated to program in several ways. All formats are accompanied with representation of example position (see Figure 1) in given format.

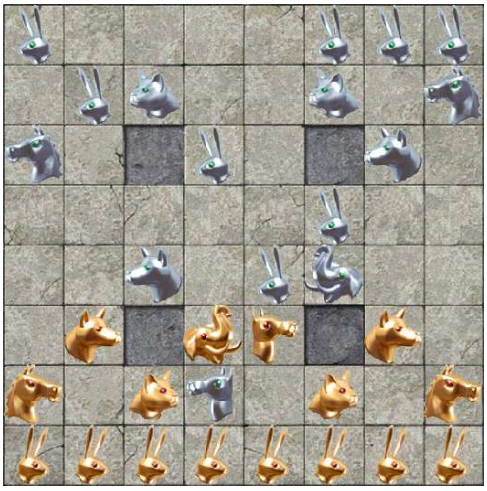

Figure 1: Example position.

standard position format

This format is used for *game_position* file in getMove mode as well as in combination with `setpositionfile` command in AEI mode. Token 8b means this is position in the 8th move with black(silver) to play.

```
8b
 +-----------------+
8| r . . . . r r r |
7| . r c . . c . h |
6| h . x r . x d . |
5| . . . . . r . . |
4| . . d . r e . . |
3| . D x E M x D . |
2| H . C m . C . H |
1| R R R R R R R R |
 +-----------------+
   a b c d e f g h
```

compact position format

Position is given by the side to move (w/b) followed by a string embraced in '[' ']' and consisting of the piece letter or a space for each of the 64 squares. This format is used only in AEI mode (in combination with `setposition` command). Information on move number is not included.

```
b[r    rrr rc  c hh  r  d      r    d re   D EM D H Cm C HRRRRRRRR]
```

game record format

This is an official system of recording the single game of Arimaa. Game recorded in this format might be passed to Akimot in getMove mode as a *game_record* file.

```
1w Ra1 Rb1 Rc1 Rd1 Re1 Rf1 Rg1 Rh1 Ha2 Db2 Cc2 Md2 Ee2 Cf2 Dg2 Hh2
1b ra8 rb8 rc8 rd8 re8 rf8 rg8 rh8 ha7 db7 cc7 ed7 me7 cf7 dg7 hh7
2w Ee2n Ee3n Ee4n Ee5n
2b ed7s ed6s ed5s rd8s
3w Ee6w me7s Db2n Dg2n
3b rc8e re8s dg7s ed4e
4w me6s Ed6e Ee6w re7s
4b db7s ee4e rb8s rd7e
5w re6e Ed6e me5w Ee6s
5b rd8s rd7s re7s rf6s
6w md5s Ee5w md4s Ed5s
6b db6e dc6s re6s dc5s
7w Ed4e dc4e Ee4s dd4e
7b ha7s de4w re5s dd4w
8w Md2e md3s Ee3w Me2n
8b
```

0.6 Arimaa Engine Interface

Arimaa Engine Interface is an interface allowing engine to connect to gameroom or other applications. *AEI* is written in Python and was contributed to the Arimaa community by Brian Haskin. AEI defines a textual protocol based on *UCI*[2] for communication with the Arimaa engine. AEI is meant to replace older *bot interface* for connecting to the online gameroom. While AEI is not yet a widely adopted communication protocol in the Arimaa programming community, it gains popularity pretty quickly. We used AEI as a primary communication protocol with Akimot and as the only mean of connecting the engine to the gameroom. Moreover we use AEI as a module in related Python applications we wrote during the development process, namely : Arimaa Test Suite, Arimaa Development GUI, Rabbit Goal Tester.

We have implemented most of the commands defined by AEI and even added some more (we call these an *extended AEI set* - for these to be recognized, engine must be started with `-e` option). The full specification of AEI protocol is given in *aei-protocol.txt* in **other/aei** directory. The following list is based on this specification and represents a list of commands supported in Akimot. Commands from extended AEI set are marked with *.

[2]Universal Chess Interface

Controller to Engine Messages

`aei` First message sent to begin the opening phase. Waits for `id` messages and an `aeiok` message back from the engine to end the opening phase.

`isready` Pings engine. Engine responds with `readyok`.

`newgame` Signals the start of a new game.

`setposition <position>` Sets the current position from the string in compact position format (see §0.5.1).

`setpositionfile*` `<filename>` Gives the path to a file with a position in standard position format.

`setoption name <id> [value <x>]` Set further options. Supported options are:
`tcmove` - The per move time for the game.
Other options (see full specification) are parsed and recognized, however they currently have no effect on the program behavior.

`makemove <move>` Makes a move. Stops any current search in progress.

`makemoverec*` Makes a move that was found as a bestmove in the last search.

`go [infinite]` Performs search according to its time management and responds with the best move found. Further option `infinite` specifies to search until the `stop` command is received.

`gonothread*` Analogical to `go` command. An engine is supposed to search in the current thread (usually current thread is used for performing the AEI communication, while search runs separately). This is useful in batch AEI scripts where commands are given sequentially in advance.

`boarddump*` Prints the current board.

`treedump*` Prints the search tree from the last search.

`eval*` Evaluates current position.

`goalcheck*` Performs goalcheck on current position.

`trapcheck*` Performs a check whether some pieces can be trapped in current position.

`stop` Stops the current search. The engine responds with the bestmove found.

`quit` Exits the session.

Engine to Controller Messages

`id <type> <value>` Sends identification during the opening phase of the session. Sends information on following identifiers `name`, `author`, `version`

`aeiok` Ends opening phase and starts the main phase of the session.

`readyok` Answer to `isready` message after all previous messages from the controller have been processed.

`bestmove <move>` Best move found in search

`info <type> <value>` Information about the current search. Akimot issues info messages of following types (all bound to previous search):

`time` How long it took to perform the search.

`winratio` Expected winratio of the suggested bestmove.

`stat` Various statistics from the previous search. These include information on number of playouts, playouts per second, number of nodes in the tree (all, expanded, pruned), average descend depth, etc.

`goalcheck` Information on performed goalcheck.

`trapcheck` Information on performed trapcheck.

`log <string>` Logging information. Log messages start with *Error:*, *Warning:* or *Debug:* to indicate special handling by the controller.

0.7 Arimaa Test Suite

Arimaa Test Suite is a small Python application for testing the strength of an Arimaa engine on predefined Arimaa positions representing particular tactical or strategic maneuvers. This tool can be viewed as a sort of unit test for algorithmic side of the program. The main motivation for creating this framework was a need for quick testing whether algorithmic changes hurt or improve the performance of the program. Even though its accuracy is questionable it proved to be a useful tool to quickly identify clearly bad extensions.

The framework has a given format for defining tests. Every test carries : single Arimaa position, a comment on position, multiple tags and test how well are the criteria fulfilled. Currently implemented criteria are: score goal, prevent opponent's goal and piece position criteria which is basically an *AND-OR* description of position changes with weights. This description (*after_piece_position* field) is a collection of *blocks* separated by "|" character. Every block consists of *atoms* separated by whitespace and optional weight indication in the end of the block definition separated by ":" character. Atoms express certain assumptions about position of pieces after the move. If the assumption is correct after the particular move we say that atom is *satisfied*. Atom can be of one of the following types:

- An indication of position - e.g. `Eb3`, this atom is satisfied if after the move there will be gold elephant at `b3`
- a negation of position indication - e.g. `!ce2`, this atom is satisfied if after the move there is not a silver cat at `e2`.
- An indication of trapping - e.g. `Hf6x`, this atom is satisfied if during the move the gold horse is trapped at `f6`.

A block is *satisfied* if all of its atoms are satisfied. If a block is satisfied it is *evaluated* with its weight (or 1.0 if there is no explicit weight given) otherwise it is evaluated with 0. Weights are meant to provide broader evaluation for possible moves than just passes/fail. The result of the test is the maximum from evaluations of its blocks.
Example ATS test file (corresponding to position depicted in Figure 2):

```
[settings]

comment =
  Elephant blockade at f7.

position =
  12b
   +-----------------+
  8| r r r r . c r . |
  7| h . c m r E h r |
  6| . d X . . X d . |
  5| . . . . . r . . |
  4| . . . . . . . . |
  3| . H C e . X D . |
  2| . D . M . C . H |
  1| R R R R R R R R |
   +-----------------+
     a b c d e f g h

tags = elephant, blockade

[criteria]

condition = piece_position

after_piece_position = re8 re7 cd7 me6 | re8 re7 md7 ce6 : 0.8 |
                       re8 re7 md7 de6 : 0.85 | re8 cd7 me7 re6 : 0.7
```

When started (`python ats.py`), *ATS* reads from its configuration file (default is *ats.cfg*) information on: time per test, tests to use, how often repeat every test, engine

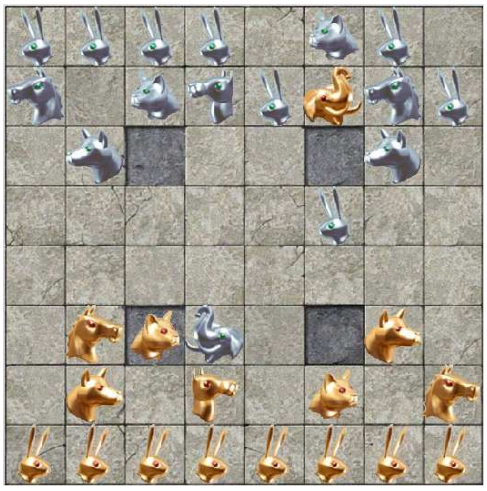

Figure 2: ATS example position.

to connect, where to log. The engine is connected through AEI and selected tests are sequentially presented to it. This is accompanied with logging information. In the end, statistics on performance are given to the user. For more examples please see tests in **akimot/other/ats/tests** directory.
Example configuration file:

```
[global]
engine = akimot/akimot -c akimot/akimot.cfg
tests = 15
time_per_test = 5
cycles = 1
```

0.8 Gameroom

There is an online gameroom (created by Omar Syed) accessible to both human players and bots at *http://arimaa.com/arimaa/gameroom/*. Akimot engine uses AEI to connect to the gameroom. A bot's account must be created online in the gameroom first. Such a bot is then identified by its name and password for connection to the gameroom. There is a script called *gameroom.py* in **other/aei** directory for connecting the bot to the gameroom. Attributes of a session (bot's name, password for connection, engine to use, number of games to play, time settings, etc.) are specified in the configuration file called *gameroom.cfg* in the same directory.

0.9 Match environment

Besides connecting the engine to the online gameroom it is naturally possible to let it play against other engines offline. There are several freely downloadable engines at Arimaa homepage for this purpose - for instance *bot_sample* by Don Dailey or *bot_fairy* by Mika Ole Hansson. There is an AEI extension for offline match as well, however the mentioned engines don't support the AEI. For this reasons we sticked with using the *match environment* based on getMove interface also created by Omar Syed. The match environment is positioned in **other/match** directory. Every bot is supposed to have its own directory (e.g. **other/match/bot_akimot**). The alias names for bots (with possibly various command line arguments) are listed in configuration file *bots*. The script conducting the game is called *match*. A game between two bots is performed by issuing `match bot_1_alias bot_2_alias` where aliases must be defined in *bots* file.

For our purposes we have created a *tour.py* script to perform a tournament between two engines consisting of specified number of games. The *tour.py* script iteratively performs the match between specified bots by calling the *match* script itself. It makes the process of organizing the results more convenient. Configuration files for bots (if available), setup notice and record of every match played in the tournament are stored in the tournament's directory. Moreover result of the tournament is written to file *list.txt* on the same level as tournament's directory.
The syntax is following: `python tour.py bot_1_alias bot_2_alias [options]`
Where options are:

`--game_dir dirname` Tournament's directory (default **matches/some_number**).

`--games_num number` Number of games in the tournament (default 100).

`--comment comment` Comment on a tournament.

`--silent` Ignore the bots log output (otherwise written to appropriate log file in the final directory).

`--mode` Running mode - default is *standalone*, other options are *master, slave, finish_only*. This option served a very specific purpose during the development and is not recommended to use. For more details see below.

For effectively running the large number of games we have devised the *tour.py* script with ability to run simultaneously on (potentially) multiple machines (aka cluster). This was quite specific demand in the development process and is meant for interested and experienced users (won't work out-of-the-box). The principle is following. The code and configuration files that the games should be run with, must be present on the cluster machines. Moreover it is advisable if the cluster machines have a shared filesystem (i.e. AFS). On the controlling machine the *tour.py* script is invoked in *master* mode. The script connects to predefined computers (via ssh using the *psshlib.py* script based on the *pexpect* library) and starts its instances in *slave* mode, these run only a single game and finish. After which the controlling script starts the next instance. In the end final log entry to the *list.txt* is made by a single instance in *finish_only* mode. The *psshlib.py*

script is included in the package. Users interested in this feature should study this script and define their login credentials (if any) in there.

0.10 Simple Arimaa Development GUI

During the process of development we strongly missed a simple-to-use Arimaa game viewer. While it is possible to view the game records online through a Java applet this was quite slow and sometimes unreliable. For this reason we have developed a very simple GUI for Arimaa developers. The GUI is written in *Python* using *Qt4*. It's capable of:

- Loading a game record and replaying the game with possibility to jump back and forth in the record.
- Loading a position from the standard position format (see §0.5.1).
- Dumping a position to the standard position format.

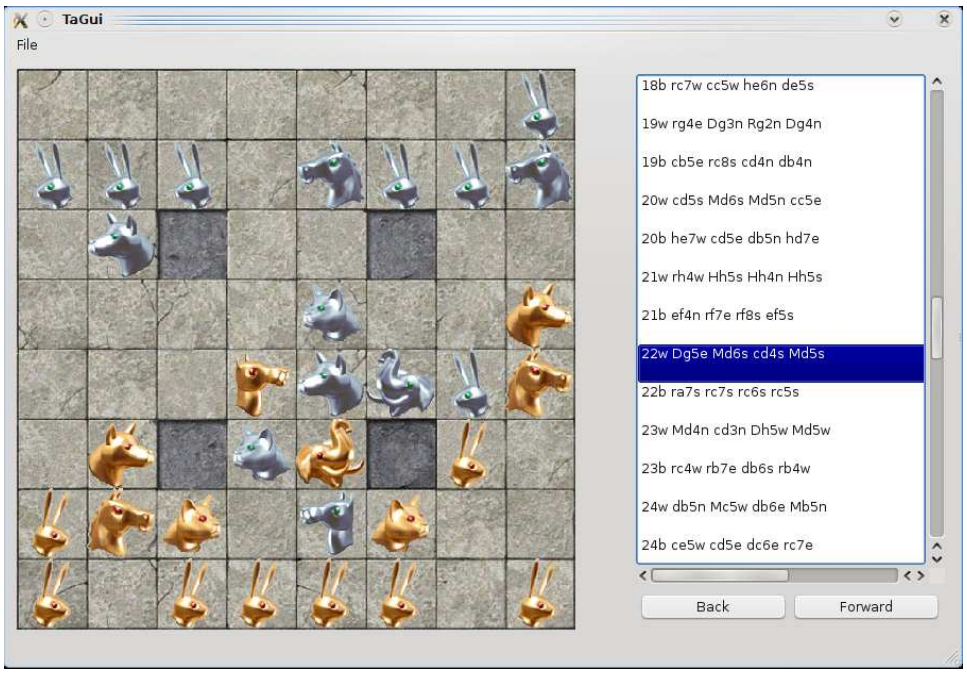

Figure 3: The development GUI.

The picture of the GUI is given in the Figure 3.